MOLE™ MICROCONTROLLER DEVELOPMENT SUPPORT

HPC™ C COMPILER USER'S MANUAL

RN

Customer Order Number 424410883-001
NSC Publication Number 424410883-001A
July 1987

MOLE™

HPC™ C Compiler User's Manual

REVISION RECORD

REVISION	RELEASE DATES	SUMMARY OF CHANGES
A	07/87	First Release. MOLE ™ HPC™ C Compiler User's Manual NSC Publication Number 424410883-001

PREFACE

This manual describes National Semiconductor's HPC™ (High Performance Controller) C compiler (CCHPC). CCHPC follows the syntax and semantics of the draft ANSI standard C language (ANSI document number X3J11/86-157).

In addition to the standard C language, the HPC C compiler supports some non-standard statement types and the ability to include assembly code in-line.

The information contained in this manual is for reference only and is subject to change without notice.

CONTENTS

Chapter 1

OVERVIEW

1.1 INTRODUCTION

The HPC™ (High Performance Controller) C Compiler (CCHPC) supports the draft ANSI standard C language as defined by the *Draft American National Standard for Information Systems — Programming Language C*, ANSI Document Number X3J11/86-157. CCHPC also supports some non-standard statement types and the ability to include assembly code in-line. This manual discusses these non-standard enhancements.

Most standard C programs can be compiled by CCHPC. Most programs compiled by CCHPC, if they do not use the extensions, can be compiled by standard C compilers.

1.2 MANUAL ORGANIZATION

Chapter 2 introduces the HPC C compiler and compiler command syntax.

Chapter 3 describes things basic to the compiler such as the character set, identifiers, constants, data types, preprocessor directives, initialization of variables, expression evaluation and in-line microassembler code.

Chapter 4 covers implementation dependent considerations such as storage classes and modifiers, declarations, statements, C stack format, using in-line microassembler code, efficiency considerations and run-time notes.

Appendix A shows how to convert between standard C and CCHPC.

Appendix B contains the invocation line syntax for the different host operating systems.

Appendix C contains a complete list of compiler error messages.

1.3 DOCUMENTATION CONVENTIONS

Italics are used for items supplied by the user. The italicized word is a generic term for the actual operand that the user enters.

Spaces or blanks, when present, are significant; they must be entered as shown. Multiple blanks or horizontal tabs may be used in place of a single blank.

{ } Large braces enclose two or more items of which one, and only one, must be used. The items are separated from each other by a logical OR sign "|."

[ ] Large brackets enclose optional item(s).

| Logical OR sign separates items of which one, and only one, may be used.

... Three consecutive periods indicate optional repetition of the preceding item(s). If a group of items can be repeated, the group is enclosed in large parentheses "( )."

,,, Three consecutive commas indicate optional repetition of the preceding item. Items must be separated by commas. If a group of items can be repeated, the group is enclosed in large parentheses "( )."

( ) Large parentheses enclose items which need to be grouped together for optional repetition. If three consecutive commas or periods follow an item, only that item may be repeated. The parentheses indicate that the group may be repeated.

␣ Indicates a space. ␣ is only used to indicate a specific number of required spaces.

All other characters or symbols appearing in the syntax must be entered as shown. Brackets, parentheses, or braces which must be entered, are smaller than the symbols used to describe the syntax. (Compare user-entered [ ], with [ ] which show optional items.)

1.3.1 Example Conventions

In interactive examples where both user input and system responses are shown, the machine output is in regular type. User-entered input is in boldface type. Output from the machine which may vary (*e.g.*, the date) is indicated with italic type.

This document contains keywords that must be entered in lower case. These keywords have been indicated by boldface type.

Chapter 2

THE HPC C COMPILER

2.1 INTRODUCTION

The HPC C Compiler (CCHPC) supports the C language with some non-standard enhancements. The enhancements include the support of two non-standard statement types (loop and switchf), non-standard storage class modifiers (discussed in Section 4.7) and the ability to include assembly code in-line. The compiler supports enumerated types, passing of structures by value, functions returning structures, function prototyping and argument checking.

2.2 COMPILER COMMAND SYNTAX

The CCHPC runs under different host operating systems. Depending on the host system, the *cchpc* command line options, ordering of the elements and their syntax may vary. See Appendix B for the command line information for your specific host. In all cases, the command line consists of command name, some options or switches and the filename to be compiled.

The compiler output, in the form of ASMHPC assembler source statements, is put in a file with the extension ".asm" replacing the extension of the source file.

The following is a list of options or switches and their description:

Including the C code in the assembly code.
: If selected, the assembly code output file will contain the C source code lines as comments. This is very useful if a person is to read the file, but slightly slows the compilation and the assembly processes.

Invoking the C preprocessor before compilation.
: Normally, the preprocessor is invoked to handle the #define and #ifdef type lines and macros. This option allows the preprocessor to be skipped.

Setting the execution stack size.
: This switch takes a numeric argument, in the form of a C-style constant. If the module contains the function *main*, the compiler uses the number as the size of the program's execution stack memory section, in words. If the module does not contain *main*, the option is ignored. If the module does contain *main* and no stack size option is given, a default value is used. The default can be seen by invoking *cchpc* with no arguments.

Creating 8-bit wide code.
: Normally, the code generated is to be run from 16-bit wide memory. This switch causes the code to be runnable from 8-bit memory by avoiding instructions (such as JIDW offsets) which require 16-bit width.

Placing string literals in ROM.
: The language standard calls for string literals to be in RAM and individual copies for each usage of the literal. Therefore, the compiler will copy the literal data from ROM to RAM on startup unless this option is requested. If

this option is requested, the strings stay in ROM and are unmodifiable and may be combined to the same instance of the data. This saves startup time, RAM space and ROM space. Note that this does not affect string variables, whose initialization values still get copied from ROM into RAM.

Turning off compiler warning messages.

Indicating directories for include files.

This switch takes a string argument, which is a file system directory name on the host system. The string argument is passed to the C preprocessor, which uses it as a directory to search for include files.

Defining symbol names.

This switch takes a string argument in one of two forms:

symbolname
symbolname=stringvalue

which is passed to the C preprocessor. These are the same as the lines in the file saying:

#define *symbolname* 1
#define *symbolname stringvalue*

at the very beginning.

Undefining symbol names.

This switch takes a string argument in the form:

symbolname

which is passed to the C preprocessor. It removes any initial definition of *symbolname*, defined by the preprocessor itself or previously defined by an option in the same invocation line. It does not affect any subsequent definition of *symbolname* in the program.

Permitting old-fashioned constructs.

Certain anachronisms from the Kernighan and Ritchie C, such as =+ for += and variable initialization without the = to indicate the value as in **int x 5** for **int x = 5**, are not permitted in ANSI C but will be accepted by the compiler if this option is used.

Debugging the compiler.

This option causes the compiler to print large amounts of information about its internal workings. This information is useful only for the support of the compiler and is not intended for users.

Chapter 3

BASIC DEFINITIONS

3.1 INTRODUCTION

CCHPC and other C compilers may differ syntactically. This chapter summarizes the syntax accepted by CCHPC, where those differences may exist. Also discussed are data types and their use.

3.2 NAMES

A name may be arbitrarily long, but only the first 32 characters are significant. Case distinctions are respected. The first character must be alphabetic or an "_" (underscore); the rest must be alphabetic, numeric or "_" (underscore).

3.3 CONSTANTS

The compiler supports the use of decimal, octal, hex, character and string constants.

A decimal constant is any string of digits, not beginning with the digit zero.

An octal constant is a string of digits 0 through 7 beginning with the digit zero.

A hex constant is a string composed of digits and the letters **a** through **f** and beginning with **0x**. For example, 0xffff, 0Xf000, and 0x001 are hex constants. Letters in a hex constant may be upper case, lower case or mixed.

Octal, decimal or hex constants may be suffixed by **l** or **L** to indicate that they are being forced to be of type "long." They may be additionally or alternatively suffixed by **u** or **U** to indicate that they are unsigned.

A floating point constant consists of an integer part, followed by a decimal point, followed by a fractional part, followed by a possible signed exponent part. Both integer and fractional parts consist of a string of decimal digits. The exponent part consists of either "E" or "e", followed by an optional sign, followed by a string of digits. Either the integer part or the fractional part, but not both, may be missing. Either the fractional part or the exponent part, but not both, may be missing. The constant is stored within the compiler as a double precision floating point constant in the format used by the host. Floating point constants may have a suffix of "f" or "F" appended, to indicate type "float" instead of "double" (which is the default). Since all floating numbers on the HPC are 32-bit, the distinction is irrelevant.

A character constant consists of a single character enclosed in unescaped single quotes. The standard C escape sequences are supported.

A string constant is a string of characters enclosed in unescaped double quotes. The compiler null-terminates a string with a '\0'. The value of a string constant is the address of the first byte of the string. Two or more adjacent string constants, separated only by whitespace (blanks, tabs, newlines, and/or form feeds), are concatenated into a single string constant. The maximum length of a string constant is 200 characters.

3.4 ESCAPE SEQUENCES

The following standard escape sequences for non-printing characters are supported:

\n	new-line (line feed)
\t	horizontal tab
\v	vertical tab
\b	back space
\r	carriage return
\f	form feed
\a	alert (audible signal - beep or bell)
\\	backslash
\'	single quote (use inside character constant)
\"	double quote (use inside string constant)
nnn	*nnn* = 1-3 digit octal number
\x*nnn*	*nnn* = 1-3 digit hexidecimal number

These characters may be used as character constants or as part of a string constant. If a backslash is followed by a newline character (*i.e.*, backslash is the last character on the line), then these characters are ignored. This allows the programmer to spread a string constant over more then one line. If a backslash is followed by any other character which is not in the list above, then the backslash is ignored.

3.5 COMMENTS

Comments begin with "/*" and end with the first "*/" which follows on the input stream. Comments cannot be nested.

3.6 DATA TYPES

The HPC C compiler supports the following data types:

NAME	SIZE IN BITS
char	8
short	16
int	16
enum	8 or 16
long	32
signed char	8
signed short	16
signed int	16
signed long	32
unsigned char	8
unsigned short	16
unsigned int	16
unsigned long	32
float	32
double	32
long double	32
struct	sum of component sizes
union	maximum of component sizes

The type "char" is treated as signed.

The keywords const and volatile can be applied to any data type. The use of const indicates the symbol refers to a location which may be only read. If the symbol is in static or global storage, it will be assigned to ROM memory. The use of volatile indicates that optimization must not change or reduce the accesses to the symbol. This permits accessing locations predictably, such as I/O registers, which have side effects.

Unsigned operations are the same as signed operations, except for multiplication, division, remainder, right shifts and comparisons. For signed integers, the compiler uses an arithmetic shift when shifting right. For unsigned integers, the compiler uses a logical shift when shifting right.

The architecture of the HPC is strongly oriented towards unsigned arithmetic, therefore unsigned variables should be used, except for cases that absolutely require signed arithmetic.

Because the HPC supports 8-bit operations, CCHPC differs from the usual practice of C compilers in that it does not automatically promote "char" types to "int" when evaluating expressions. When generating code for a binary operation, the compiler will promote a "char" operand to "int" only if the other operand is a 16-bit (or more) value or if the result of the operation is required to be a 16-bit (or more) value. The use of 8-bit operations yields efficient code without compromising the correctness of the result.

Bit fields must be declared as int, signed int or unsigned int. The default for an int bit field is unsigned. Bit fields may be signed or unsigned. A bit field cannot be inside a char. Signed bit fields are extended when extracted. However, the compiler can store a bit field in an int or a char. The maximum size of a bit field is 16-bits. Bit fields are assigned starting at the least significant bit in the byte or word. Bit fields can be set up in structures in either 8- or 16-bit types.

Access to most bit fields is usually expensive. For instance, extracting a bit field involves a shift and a bitwise AND operation. Storing a value into a bit field is even more expensive (*i.e.*, it takes two loads, a store, two AND's, an OR, a shift, a push and a pop). However, if a bit field is one bit wide it can be tested or set to constant values efficiently.

3.7 PREPROCESSOR DIRECTIVES

The HPC C compiler uses the standard C preprocessor, therefore, any of the preprocessor functions, including "#define", "#include" and macros with arguments, can be used.

3.8 PROGRAM ORGANIZATION

A program is a set of intermixed variable and function definitions. A variable must always be defined before its first use. Functions may be defined in any order.

3.9 INITIALIZATION OF VARIABLES

Variables may be initialized when they are declared, according to the draft ANSI standard rules. Initialization of automatic variables is done as the program is running. Initialization of external or static variables is done when program execution starts.

3.10 OPERATORS

The hierarchy of operators, from lowest precedence to highest, is as follows:

```
,
=  +=  -=  /=  *=  %=  <<=  >>=  &=  |=  ^=
?: (Conditional)
||
&&
|
^
&
==  !=
<  <=  >  >=
<<  >>
+  -
/  *  %
unary  ++  --  -    !  &  *  sizeof  (cast)(expr)
->  .  function calls subscripting
(expr) name constant
```

The comparison operators generate a zero, if false, or a one, if true.

The right shift is a logical shift if the left operand is unsigned. Otherwise, it is an arithmetic shift.

Structure assignment is supported, along with the passing of structures by value as function arguments and the returning of structures from functions. The only other things that can be done with a structure or union identifier are to take its address or to select a member using the "." operator.

In general, errors such as arithmetic overflow or out-of-bounds addresses go undetected and have undefined results.

3.11 IN-LINE MICROASSEMBLER CODE

It is possible for the programmer to enter directly into assembly language simply by entering a "/$". All the data following the "/$" is copied to the assembler output file until the compiler sees a terminating "$/", which ends the assembler inclusion. This may be done a line at a time, as in:

```
/$ microassembler line $/
```

or over several lines, as in:

```
/$
assembler line..
assembler line..
 $/
```

The information between "/$" and "$/" is always placed in the same memory as compiler-generated code. Section 4.6 describes the use of in-line microassembler code.

Chapter 4

IMPLEMENTATION DEPENDENT CONSIDERATIONS

4.1 INTRODUCTION

This chapter discusses implementation-dependent considerations, such as memory, storage classes, C stack format and using in-line microassembler code.

4.2 MEMORY

Code generated by the compiler is intended to be run from 16-bit wide memory. If the system design calls for 8-bit wide ROM memory bus, then the code should be compiled with the 8-bit wide code switch. The compiler will then generate code, which avoids using the JIDW instruction for a switch statement.

4.3 STORAGE CLASSES

CCHPC supports the following storage classes:

auto
static
register
typedef
extern

Because the HPC processor has few registers, the "register" keyword is always ignored. Instead, the declared variables will be treated as "auto" (unless the NOLOCAL storage class modifier is in effect, in which case they will be treated as "static"). For efficient access to a variable, use the BASEPAGE storage class modifier, instead of "register."

In global declarations, the default storage class is "static."

In declarations within functions, the default storage class is "auto."

4.3.1 Storage Class Modifiers

To support certain machine-dependent features of the HPC architecture, the compiler supports the notion of the "storage class modifier." Syntactically, a storage class modifier may appear with or in place of a storage class. The following storage class modifiers are supported:

KEYWORD	APPLICABLE TO
BASEPAGE	variable
ACTIVE	function
NOLOCAL	function
INTERRUPT1	function
INTERRUPT2	function
INTERRUPT3	function
INTERRUPT4	function
INTERRUPT5	function
INTERRUPT6	function
INTERRUPT7	function

These keywords must be entered in upper case as shown. Zero or more storage class modifiers can be supplied with each variable or function declaration. The compiler will generate an error message if it finds a conflicting use of storage class modifiers (such as INTERRUPT1 ACTIVE).

The effect of each keyword is as follows:

BASEPAGE
: The variable will be allocated in the BASE (base page) section. Since accessing a base page variable is more efficient than accessing any other type of variable and since the amount of the base page storage is limited, great care should be taken when deciding which variables should have the BASEPAGE modifier.

ACTIVE
: The address of the function will be placed in one of the entries of the 16 word JSRP table. Subsequent calls to the function will occupy a single byte, instead of two or three. Any function which will be called frequently should be considered for designation as an ACTIVE function. At most 16 functions can be designated as ACTIVE. If a program uses an indirect function call, then one of the 16 slots is reserved for an indirect call routine. In order to obtain the full savings of space and time, an ACTIVE function should be defined before it is first used.

NOLOCAL
: This means that the function's local variables are not on the run-time stack. Instead, declared variables are allocated in static storage. If the function is called recursively then any new invocation will use the same local variables as the last invocation. Offsetting this disadvantage is the fact that access to local variables in a NOLOCAL function will be much more efficient. Furthermore, if the function is defined to have no arguments, then entry to and exit from the function will be much more efficient, because there is no need to adjust the frame pointer on entry and exit.

INTERRUPT1
INTERRUPT2
INTERRUPT3
INTERRUPT4
INTERRUPT5
INTERRUPT6
INTERRUPT7

These modifiers can be used to set interrupt vectors (one through seven) to point to a particular function. A given function may be associated with more than one interrupt. A given interrupt number may be only applied to one function. Any function which has an INTERRUPT storage class modifier has a special entry and exit code generated for it. The entry code pushes all the registers (A, B, X, K, and PSW) and word at 0 onto the stack before executing the normal function entry code. The exit code restores the word at 0 and all the registers which were saved and executes a return from interrupt instruction.

4.4 C STACK FORMAT

The Stack Pointer (SP) starts at the start address assigned by the linker and moves towards successively higher locations. The Stack Pointer always points at the next free location at the top of the stack.

Within a function, the compiler maintains a Frame Pointer (FP), which it uses to access function arguments and local automatic variables. The highest word location in base page memory (0xbe) is reserved by the compiler to hold the Frame Pointer.

To call a function, the compiler pushes arguments onto the stack in reverse order, (the PUSH instruction increments the SP by 2 each time it is executed), calls the function, then decrements the Stack Pointer by the number of bytes pushed. For instance, to call a function with two one-word arguments, the compiler would emit code to do the following:

```
PUSH        arg2            (SP += 2)
PUSH        arg1            (SP += 2)
jump subroutine to function
SUB         SP,4            (SP -= 4)
```

The jump subroutine instruction pushes the current program counter onto the stack. Because all stack pushes are 16-bit pushes, any 8-bit function argument is automatically promoted to 16 bits.

On function entry, the compiler creates new stack and frame pointers by computing:

PUSH FP
FP = SP
SP = SP + *framesize*;

where *framesize* is the space required for all local automatic variables. If the frame size is odd, the compiler always rounds it up to the next even number, in order to avoid a Stack Pointer with an odd address. If there are two arguments and two local variables, then the frame size would be 4 and the stack would look like this:

FP-8	second argument
FP-6	first argument
FP-4	return address
FP-2	old FP
FP+0	first local variable
FP+2	second local variable
FP+4	next free stack location (same as SP)

If a function argument is defined to be an 8-bit type, then only the lower eight bits of the value pushed by the caller will be referred to inside the called function.

On function exit, the compiler restores the SP and FP to the way they were on entry by executing the following:

```
SP = FP
POP FP
RET
```

The return instruction (RET) sets the new program counter by popping the saved program counter off the stack.

4.5 USING IN-LINE MICROASSEMBLER CODE

Assembler code should be entered in the body of a function beginning with a "/$" and ending with a "$/". When this is done, the programmer will need to be able to relate the code to variables previously declared.

Any of the currently active variables can be accessed by entering:

@*name*

where *name* is the name of the variable.

For example, an included assembler line to move any variable "alpha" to any variable "beta" would look like this:

```
/$
    LD    A,@alpha
    ST    A,@beta
$/
```

If neither variable is an argument or has automatic storage class, then the variable move could be written as

```
/$
      LD      @beta,@alpha
$/
```

The *@name* is replaced by one of the following assembler expressions for representing the *value* of the variable.

STORAGE CLASS	**@NAME REPLACED BY**
extern	_name
global	_name
static	Vn.t
argument	offset[FP].t
automatic	offset[FP].t

where _name is the original source name prefixed with "_ ", t is either **B** (if the type is 8-bit) or **W** (otherwise), V*n* is a name generated by the compiler (and "*n*" is a decimal number), and "offset" is a hex offset into the current stack frame. Section 4.4 describes the stack format and explains the significance of the Frame Pointer (FP).

When a variable's storage class is static, "@variable" can be used in any context where a "direct address" is permitted by the assembler. When a variable is a function argument or has automatic storage class, only "@variable" can be used in a context where an "indexed" operand address is permitted.

To get the *address* of a variable in assembler, use:

@^*name*

This will be replaced according to the storage class as follows:

STORAGE CLASS	**@^NAME REPLACED BY**
extern	_name
global	_name
static	Vn
argument	offset
automatic	offset

where _name is the original source name prefixed with "_ ", V*n* is a name generated by the compiler (and "*n*" is a decimal number) and "offset" is a hex offset into the current stack frame.

If the storage class is static, the address in the A register can be set using:

```
LD          A,@^name
```

If the storage class is automatic or has an argument, then the following must be used:

```
LD      A,FP
ADD     A,@^name
```

On entry to an in-line assembler code section, the compiler guarantees that it will not have any active registers.

The A, B, K, and X registers can be used at any time. The word at address 0 is available as a temporary scratch location. The compiler and library use it, but never to retain a value across commands. The user may use it similarly.

If the FP must be adjusted, then it must be restored before exiting. However, if a compiler-declared variable "*name*" is being referenced using @*name*, then the FP must not be modified in any way.

Because the compiler handles all storage allocation, any storage required by in-line assembler code must be declared in C code, then referred to using the conventions previously described.

4.6 EFFICIENCY CONSIDERATIONS

Not all variables are created equal. The best way to reduce code size and improve running time is to use lots of BASEPAGE variables. BASEPAGE variables are the most efficient because they can use an 8-bit direct address. They are also especially efficient for holding pointers and structure pointers. The second choice would be to use a static variable, which uses a 16-bit direct address. The least efficient variable to access is an automatic variable on the stack, which uses an indirection through an eight-bit address indexed by an 8- or 16-bit offset. To maximize the use of 8-bit offsets in automatic variable accesses, make sure to declare smaller objects (characters and integers) before larger objects (arrays and structures).

In order to save space and time, avoid the use of long integers and floats, except where absolutely necessary.

Because the architecture of the HPC is strongly oriented towards unsigned arithmetic, the programmer should use unsigned variables except for cases that absolutely require signed arithmetic.

Unsigned comparisons for ">=" or "<=" are more efficient than for ">" or "< ". Signed comparisons are less efficient than any unsigned comparisons.

Since the compiler does not attempt to identify common subexpressions and arrange that they be computed only once, it should be done by the programmer. For example, if a program contains the following roots of a quadratic equation:

```
if( (B*B - 4*A*C) > 0 )
{
        printf("Real roots %f and %f\n",
                (-B - sqrt(B*B - 4*A*C)) / (2*A),
                (-B + sqrt(B*B - 4*A*C)) / (2*A));
}
else
{
        printf("Imaginary: (%f,%fi) and (%f,%fi)\n",
                -B / (2*A), - sqrt(-(B*B - 4*A*C)) / (2*A),
                -B / (2*A), + sqrt(-(B*B - 4*A*C)) / (2*A));
}
```

the compiler will not recognize the multiple uses of "B*B - 4*A*C" or "2*A" as common subexpressions. For each occurrence, the compiler will generate code resulting in 5 and 6 computations of these values.

To localize the calculation to one place and one time for each equation, the programmer might want to declare a local variable, evaluate the common expression into it, and then use the local variable in place of the expression thereafter. For example, the previous equations can be coded as follows:

```
discr = B*B - 4*A*C;
denom = 2*A;
if( discr > 0 )
{
        discr = sqrt(discr);
        printf("Real roots %f and %f\n",
                (-B - discr) / denom,
                (-B + discr) / denom);
}
else
{
        discr = sqrt(-discr);
        printf("Imaginary: (%f,%fi) and (%f,%fi)\n",
                -B / denom, - discr / denom,
                -B / denom, + discr / denom);
}
```

4.6.1 Declaration Syntax

CCHPC supports the draft ANSI standard syntax for declarations by allowing the programmer to define pointers to, arrays of, or functions returning arbitrary objects.

Be aware that the way a data structure is defined will have a bearing on the efficiency of the compiled program. In the case of array subscripting, C syntax requires that the subscript be multiplied by the size of the object before being added to the pointer of the array of objects. If the size of the object is not one, then the compiler will have to generate a multiplication. If the size of the object is a power of 2, then the multiplication may be converted to a shift.

This means that it may be expensive to use an array of an array, a structure, or a union.

For instance, with the following program:

```
struct{
    int x;
    int y;
    int z;
} points[10];
```

the compiler would have to convert the array of structure reference from:

```
points[i].y = 2;
```

into:

```
*(&points + (i*6) + 2) = 2;
```

This conversion would require a multiplication. On the other hand, suppose the program had:

```
int xpts[10];
int ypts[10];
int zpts[10];
```

Then the programmer would code:

```
xpts[i] = 2;
```

which the compiler would convert to:

```
*(&xpts + i*2) = 2;
```

Since the multiplication by two is converted to a 1-bit shift by the compiler, no multiplication is required.

4.7 STATEMENTS AND IMPLEMENTATION

S, S1, S2, S*n* are statements. The keywords described in this section must be entered in lower case. In the following statements, these keywords are indicated in boldface type and the punctuation required is shown:

expression;
if(*expression*) *S*
if(*expression*) *S1* **else** *S2*
while(*expression*) *S*
do *S* **while** (*expression*);
for(*e1*; *e2*; *e3*) *S*
break;
goto *label*;
continue;
return;
return *expression*;
case *const-expr*:
default:
switch (*expression*) *S*
switchf (*expression*) *S*
loop (*expression*) *S*
{ *S1 S2 ... Sn* }

The switch statement will generate a jump table for a set of cases if the maximum case minus the minimum case plus one divided by the number of cases is less than 1.25. Otherwise, it simply arranges to emit code which tests for each possible value, in the order in which they appeared. The jump table type of switch is most efficient in both space and time.

The switchf statement is the same as the switch, except that, if a jump table is generated, the compiler does not generate code to check the bounds of the value being jumped on. It assumes that the value of the switchf expression will invariably select one of the cases and that the default will never be selected. It is therefore up to the programmer to ensure or enforce that the value being switched on is in range.

The loop statement is not in standard C, but has been introduced to allow the programmer to save code space in tight loops by using the HPC's DECSZ (decrement and skip if zero) instruction to control looping. The loop statement executes the statement S the number of times given by the expression. The result of the expression is treated as an unsigned integer. Because the loop counter is decremented before being tested, an expression with a value of zero will cause the loop to be repeated 65536 times. If a count larger than 65635 is given, the actual count executed will be the given count modulo 65636.

A continue statement inside a loop statement will behave the same way as if the last statement in the loop had just been executed. That is, it will decrement and test the count; then either it will branch to the top of the loop if the result of the test indicates looping should continue, or it will exit the loop.

A break statement inside a loop statement will cause an immediate exit from the loop.

A loop statement may be nested.

4.8 RUN-TIME NOTES

During evaluation of complex expressions, the compiler uses the stack to store intermediate values.

All HPC C programs begin by calling the function "main" with no arguments.

Before calling "main," run-time start-up code initializes RAM memory. The initial values of static or global variables with initialization are stored in ROM and copied to the appropriate locations in RAM. Static or global variables which are not initialized are cleared to zero.

When "main" returns to the run-time start-up routine, it executes the HALT macro. As provided, the HALT macro contains "JP ." which puts the chip in an infinite loop.

If "main" is not defined, the compiler will complain.

No check stack

Since the run-time stack is of fixed size and there is no check for stack overflow, it is up to the programmer to ensure the stack is large enough so that stack overflow does not happen.

NOTE: Memory location zero is used by the compiler and library as a scratch pad.

Appendix A

CONVERTING BETWEEN STANDARD C AND CCHPC

A programmer may want to compile their C program using the standard C compiler and run it under a UNIX® system.

This appendix explains how to set up a C program which gives the programmer the flexibility of compiling with either the HPC C compiler (CCHPC) or the standard C compiler.

To be able to easily switch between compiling a program with C and CCHPC, set up the following at the beginning of the program:

```
#ifdef REGULARC
#define switchf switch
#define loop(x) for(iiii=0;iiii<x;++iiii)
short iiii;
#define BASEPAGE
#define NOLOCAL
#define ACTIVE
#define INTERRUPT1
#define INTERRUPT2
#define INTERRUPT3
#define INTERRUPT4
#define INTERRUPT5
#define INTERRUPT6
#define INTERRUPT7
#endif
```

To compile a program using a non-HPC C compiler, use the command line option "-DREGULARC".

If using nested loop statements, it is necessary to set up a more elaborate way of redefining "loop(x)". If there are at most 3 levels of nesting, define the following:

```
#ifdef REGULARC
short iii_1, iii_2, iii_3;
#define loop1( x ) for(iii_1=0;iii_1<x;++iii_1)
#define loop2( x ) for(iii_2=0;iii_2<x;++iii_2)
#define loop3( x ) for(iii_3=0;iii_3<x;++iii_3)
#else
#define loop1( x ) loop(x)
#define loop2( x ) loop(x)
#define loop3( x ) loop(x)
#endif
```

and use "loop1," "loop2," or "loop3" as required instead of "loop."

Appendix B

INVOCATION LINE SYNTAX

B.1 INTRODUCTION

This appendix contains the invocation line syntax for the MS-DOS™, VAX™/VMS™ and UNIX operating systems.

B.2 MS-DOS

For the MS-DOS operating system, the invocation line has the following syntax:

cchpc [*options*] *filename.c* [*options*]

The compiler options may be entered before or after the filename. The default filename extension is ".c". The compiler output, in the form of assembler source statements, will be in *filename.asm* where the ".c" extension is replaced by ".asm".

The following are the compiler options:

/Csource
: Include the C code in the assembly code.

/Preprocess
: Do NOT invoke the C preprocessor before compilation.

/Stack=*number*
: Set the execution stack size to *number*.

/8bit_code
: Create 8-bit wide code.

/Romstrings
: Place string literals in ROM.

/Warnings
: Turns off compiler warning messages.

/Include *directory*
: Indicate directory to search for include files.

/Define *symbol*
/Define *symbol=val*
: Define symbol names.

/Udefine *symbol*
: Undefine symbol names.

/Oldfashioned
: Permit old-fashioned constructs.

/Z Debugging the compiler.

The required input for an option is indicated by the upper-case letter of the option name. If the entire option name is entered, it must be spelled correctly. It may be entered in upper or lower case. Do not pass an argument containing an equal sign through a batch file, the equal sign is interpreted as a space.

On the invocation line, @*filename* will read the named file and use the contents as if it were part of the invocation line. The default extension is ".cmd" and there is no whitespace between the "@" and the filename. The contents of *filename* may be on multiple lines, and each new line will be equivalent to a space on the invocation line. The files may not be nested.

B.3 VAX/VMS

For the VAX/VMS operating system, the invocation line has the following syntax:

cchpc [*options*] *filename.c* [*options*]

The compiler options may be entered before or after the filename. The default filename extension is ".c". The compiler output, in the form of assembler source statements, will be in *filename.asm* where the ".c" extension is replaced by ".asm".

The following are the compiler options:

/CSOURCE
Include the C code in the assembly code.

/NOCSOURCE
Do NOT include the C code in the assembly code.

/PREPROCESS
Invoke the C preprocessor before compilation.

/NOPREPROCESS
Do NOT invoke the C preprocessor before compilation.

/STACK=*number*
Set the execution stack size to number.

8BIT_CODE
Create 8-bit wide code.

/NO8BIT_CODE
Do NOT create 8-bit wide code.

/ROMSTRINGS
Place string literals in ROM.

/NOROMSTRINGS
Do NOT place string literals in ROM.

/WARNINGS
Print compiler warning messages.

/NOWARNINGS
Do NOT print compiler warning messages.

/INCLUDE=*directory*
Indicate directory to search for include files.

/DEFINE=*symbol*
/DEFINE=*symbol=val*
Define symbol names.

/NODEFINE=*symbol*
Undefine symbol names.

/UNDEFINE=*symbol*
Undefine symbol names.

/OLD-FASHIONED
Permit old-fashioned constructs.

/NOOLD_FASHIONED
Do NOT permit old-fashioned constructs.

/Z Debugging the compiler.

The parsing of the command is handled by the DEC™ command line interpreter according to its rules. An option name can be abbreviated with four letters (or less if unique.) If the entire option name is entered, it must be spelled out correctly. If the option is a negated switch, beginning with NO, the count does not include the NO. It may be entered in upper or lower case.

The invocation line will accept the @*filename* for substitution from a file. This is processed by the DEC command line interpreter. For details, refer to the *VAX/VMS Guide to Using DCL and Command Procedures.*

B.4 UNIX

For the UNIX operating system, the invocation line has the following syntax:

cchpc [*options*] *filename.c*

The compiler options must be entered before the filename. There is no default filename extension. The ".c", or whatever extension is used, must be given. The compiler output, in the form of assembler source statements, will be in *filename.asm* where the extension (if any) is replaced by ".asm".

The following are the compiler options:

—c Include the C code in the assembly code.

—p Do NOT invoke the C preprocessor before compilation.

—s *number*
Set the execution stack size to *number*.

—8 Create 8-bit wide code.

—r Place string literals in ROM.

—w Turn off compiler warning messages.

—I *directory*
Indicate directory to search for include files.

—D *symbol*
—D *symbol=val*
Define symbol names.

—U *symbol*
Undefine symbol names.

—o Permit old-fashioned constructs.

—z Debugging the compiler.

The UNIX command conforms to the System V interface definition. Only a single letter is used and must be the indicated case, separated from any argument by whitespace.

The feature of file substitution can be handled by use of the shell's command substitution capability. Refer to the *System V User's Guide*, Shell Tutorial.

Appendix C

COMPILER ERROR MESSAGES

name is not a label
name is not a member of a structure
name is not an argument
name is repeated in the argument list
name undefined
Arguments of *name* redefined
Array size must be an integer constant
Assignment of different structures
Assignment of pointer to non-pointer is not allowed
Auto variables treated as static in NOLOCAL function
BASEPAGE not applicable to function definition
Bit field type must be "int", "signed init", or "unsigned int"
Bit field won't fit!
Can't take address of a bit field
Can't take address of register variable
Cannot call NOLOCAL function recursively
Cannot get space for temp entry
Cannot have "declaration-list" with "parameter-type-list"
Cannot have function initializer
Cannot initialize *name* here
Cannot initialize automatic aggregate
Cannot initialize global registers
Cannot initialize typedef!!
Cannot modify "const" storage
Cannot open *filename*
Cannot take address of built-in
Cannot take address of register variable
Cannot take size of function
Cannot use 'void' here - 'int' substituted
Case constant must have integral type
Cast expression must have scalar type
Cast type must be scalar type
Character constant too long
Character string too long (> *number* characters)
Compound statement required
Constant expression required
Constant for shift or rotate < 0 or > *constant*
Constant word address is not even
Declaration of void variable ignored
Default not in switch
Division by zero
Duplicate case (*number*) in switch

EOF reading character constant
EOF reading string or character constant
Error in format of floating point constant
Expected ')'
Expected ',' or ')'
Expected ',' or ';'
Expected ',' or ';' - skipping to next ';' or '*name*'
Expected ',' or '}' - '}'
Expected ':'
Expected ':' after label
Expected '<'
Expected '>'
Expected '}'
Expected "while"
Expected constant expression
Expected constant expression after ':'
Expected identifier
Expected label name
Expected name
Expected name following @
Expression syntax error
External *name* redefined
Floating point constant must be decimal
Function *name* redefined
Function may not return array or function
INTERRUPTn conflicts with ACTIVE
Identifier list must be empty
Illegal assignment
Illegal break
Illegal character *number* (hex)
Illegal context for label *name*
Illegal context for type name *name*
Illegal continue
Illegal indirection
Illegal pointer combination
Illegal pointer operation
Illegal storage class for argument
Illegal struct/union argument
Illegal structure usage
Illegal use of built-in name
Interrupt function may not have arguments
Label *name* redefined
Left operand of '-' must have arithmetic type
Left operand of '-' must have scalar type
Left operand of '~' must have integral type
Left operand of bitwise op must have integral type
Left operand of bitwise op= must have integral type
Left operand of shift must have integral type

Left operand of shift= must have integral type
Local functions not allowed
Malloc() denied space for *string*
Maximum frame size (*constant*) exceeded
Member name required here
Missing closing brace
Missing enum definition
Missing structure definition
Missing union definition
Multiple defaults
No function arguments allowed here
No more registers available for assignment - B reused
No name given for argument # *number*
No operations defined for void type
Not a function
Not enough function arguments
Not in switch
Null character constant
Only "register" storage class permitted here
Operands of '*character*' have incompatible types
Operation has incompatible operands
Redeclaration of *name*
Remainder of division by zero
Right operand of bitwise op must have integral type
Right operand of bitwise op= must have integral type
Right operand of shift must have integral type
Right operand of shift= must have integral type
Sorry, *name* is not allowed
Sorry, bit field operations not supported
Sorry, but no procedure arguments allowed here
Sorry, floats are not supported - treated as long
Sorry, static/external initialization is not supported
Statement syntax error
Storage class modifier *name* is not allowed here
Storage class modifier of *name* redefined
Struct/union not allowed here
Structured statements nested too deeply
Switch expression must have integral type
Syntax error in type specifier
Too many arguments
Too many cases in switch
Too many function arguments
Too many initializers
Too many storage class keywords in declaration
Too many workfiles
Type attributes of *name* redefined
Type cannot be both signed and unsigned
Type of *name* redefined

Unexpected eof inside /$ ... $/
Unknown size
Unknown tag
Variable *name* undefined
Warning: & before array or function name ignored
Warning: Ambiguous assignment - '&=' assumed
Warning: Ambiguous assignment - '*=' assumed
Warning: Ambiguous assignment - '+=' assumed
Warning: Ambiguous assignment - '-=' assumed
Warning: Array has zero size
Warning: Auto variables treated as static in NOLOCAL function
Warning: Constant address may not fit into 8 bits
Warning: Constant truncated
Warning: Declaration of %s hides argument of same name
Warning: Different pointer types in conditional
Warning: Division by zero is undefined
Warning: Found ".." - assuming "..." wanted
Warning: GOTOs in to or out of LOOP statements give undefined results
Warning: Hex character constant truncated
Warning: INTERRUPT function is not intended to be called directly
Warning: Improper combination of pointer and arithmetic type
Warning: Improper combination of pointer and integer op *string*
Warning: Improper member use: *name*
Warning: Improper pointer combination
Warning: Incompatible pointer combination
Warning: Negative array size - forced positive
Warning: Octal constant truncated
Warning: Old-fashioned initialization
Warning: Stack=*number* processed only if main defined
Warning: Struct or union pointer wanted
Warning: Struct or union wanted
Warning: Switchf will never select default case
Warning: Unescaped newline in string or character constant
Warning: Unknown size
Warning: Zero array size - set to 1
Warning: "const" variable *name* should be initialized
You must declare global registers before any functions
Zero length named bit field?
\"Lvalue\" required here
"..." must be the last entry in the argument list

INDEX

P

R

S

T

U

V

National
Semiconductor

MICROCOMPUTER
SYSTEMS DIVISION

READER'S COMMENT FORM

In the interest of improving our documentation, National Semiconductor invites your comments on this manual.

Please restrict your comments to the documentation. Technical Support may be contacted at:

(800) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada only

Please rate this document according to the following categories. Include your comments below.

	EXCELLENT	GOOD	ADEQUATE	FAIR	POOR
Readability (style)	☐	☐	☐	☐	☐
Technical Accuracy	☐	☐	☐	☐	☐
Fulfills Needs	☐	☐	☐	☐	☐
Organization	☐	☐	☐	☐	☐
Presentation (format)	☐	☐	☐	☐	☐
Depth of Coverage	☐	☐	☐	☐	☐
Overall Quality	☐	☐	☐	☐	☐

NAME ______________________________ DATE ____________

TITLE ______________________________

COMPANY NAME/DEPARTMENT ______________________________

ADDRESS ______________________________

CITY ____________________ STATE ____________ ZIP ________

Do you require a response? ☐ Yes ☐ No PHONE ____________________

Comments:

____

____

____

____

____

____

FOLD, STAPLE, AND MAIL

424410883-001A

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

National Semiconductor Corporation
Microcomputer Systems Division
Technical Publications Dept. 8278, M/S 7C261
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-9968

National Semiconductor Corporation
Microcomputer Systems Division

National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051
Tel: (408) 721-5000
TWX: (910) 339-9240

National Semiconductor
5955 Airport Road
Suite 206
Mississauga, Ontario
L4V1R9 Canada
Tel: (416) 678-2920
TWX: 610-492-8863

Electronica NSC de Mexico SA
Hegel No. 153-204
Mexico 5 D.F. Mexico
Tel: (905) 531-1689, 531-0569,
531-8204
Telex: 017-73550

NS Electronics Do Brasil
Avda Brigadeiro Farla Lima 830
8 Andar
01452 Sao Paulo, Brasil
Telex: 1121008 CABINE SAO PAULO
113193 INSBR BR

National Semiconductor GmbH
Furstenriederstraße Nr. 5
D-8000 München 21
West Germany
Tel.: (089) 5 60 12-0
Telex: 522772

National Semiconductor (UK), Ltd.
301 Harpur Centre
Horne Lane
Bedford MK40 1TR
United Kingdom
Tel: 0234-47147
Telex: 826 209

National Semiconductor Benelux
Ave. Charles Quint 545
B-1080 Bruxelles
Belgium
Tel: (02) 4661807
Telex: 61007

National Semiconductor (UK), Ltd.
1, Bianco Lunos Allè
DK-1868 Copenhagen V
Denmark
Tel: (01) 213211
Telex: 15179

National Semiconductor
Expansion 10000
28, Rue de la Redoute
F-92 260 Fontenay-aux-Roses
France
Tel: (01) 660-8140
Telex: 250956

National Semiconductor S.p.A.
Via Solferino 19
20121 Milano
Italy
Tel: (02) 345-2046/7/8/9
Telex: 332835

National Semiconductor AB
Box 2016
Stensätravägen 4/11 TR
S-12702 Skärholmen
Sweden
Tel: (08) 970190
Telex: 10731

National Semiconductor
Calle Nunez Morgado 9
(Esc. Dcha. 1-A)
E-Madrid 16
Spain
Tel: (01) 733-2954/733-2958
Telex: 46133

National Semiconductor Switzerland
Alte Winterthurerstrasse 53
Postfach 567
CH-8304 Wallisellen-Zürich
Tel: (01) 830-2727
Telex: 59000

National Semiconductor
Pasilanraitio 6C
SF-00240 Helsinki 24
Finland
Tel: (90) 14 03 44
Telex: 124854

NS Japan K.K.
POB 4152 Shinjuku Center Building
1-25-1 Nishishinjuku, Shinjuku-ku
Tokyo 160, Japan
Tel: (03) 349-0811
TWX: 232-2015 NSCJ-J

National Semiconductor Hong Kong, Ltd.
1st Floor,
Cheung Kong Electronic Bldg.
4 Hing Yip Street
Kwun Tong
Kowloon, Hong Kong
Tel: 3-899235
Telex: 43866 NSEHK HX
Cable: NATSEMI HX

NS Electronics Pty. Ltd.
Cnr. Stud Rd. & Mtn. Highway
Bayswater, Victoria 3153
Australia
Tel: 03-729-6333
Telex: AA32096

National Semiconductor PTE, Ltd.
10th Floor
Pub Building, Devonshire Wing
Somerset Road
Singapore 0923
Tel: 652 700047
Telex: NATSEMI RS 21402

National Semiconductor Far East, Ltd.
Taiwan Branch
P.O. Box 68-332 Taipei
3rd Floor, Apollo Bldg.
No. 218-7 Chung Hsiao E. Rd.
Sec. 4 Taipei Taiwan R.O.C.
Tel: 7310393-4, 7310465-6
Telex: 22837 NSTW
Cable: NSTW TAIPEI

National Semiconductor (HK) Ltd.
Korea Liaison Office
6th Floor, Kunwon Bldg.
No. 2, 1-GA Mookjung-Dong
Choong-Ku, Seoul, Korea
C.P.O. Box 7941 Seoul
Tel: 267-9473
Telex: K24942